GIT

1. Git 的身世

作者：林纳斯·托瓦兹 （Linus Torvalds），Linux 的伟大的副产物

Linus 在 1991 年创建了开源的 Linux 之后靠着开发者共同维护。

2002 以前，contributors 把源代码文件通过 diff 的方式发给 Linus，Linus 和 维护者 `手工方式` merge。

维护者受不了了，Linus 选择了 BitKeeper，很喜欢 BitKeeper.

理查德·斯托尔曼（Richard Stallman）自由软件倡导者，精神领袖，GNU计划创造者


1:26 / 2:21


1:55 / 2:21

并且有人开始对 BitKeeper 逆向，破解，BitKeeper 收回了 Linus 的免费使用权。

不得不要写一个自己的版本控制系统：

- 1.速度优势，有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）
- 2.对非线性开发模式的强力支持（允许上千个并行开发的分支）
- 3.完全分布式
- 4.简单易用的设计，bullshit

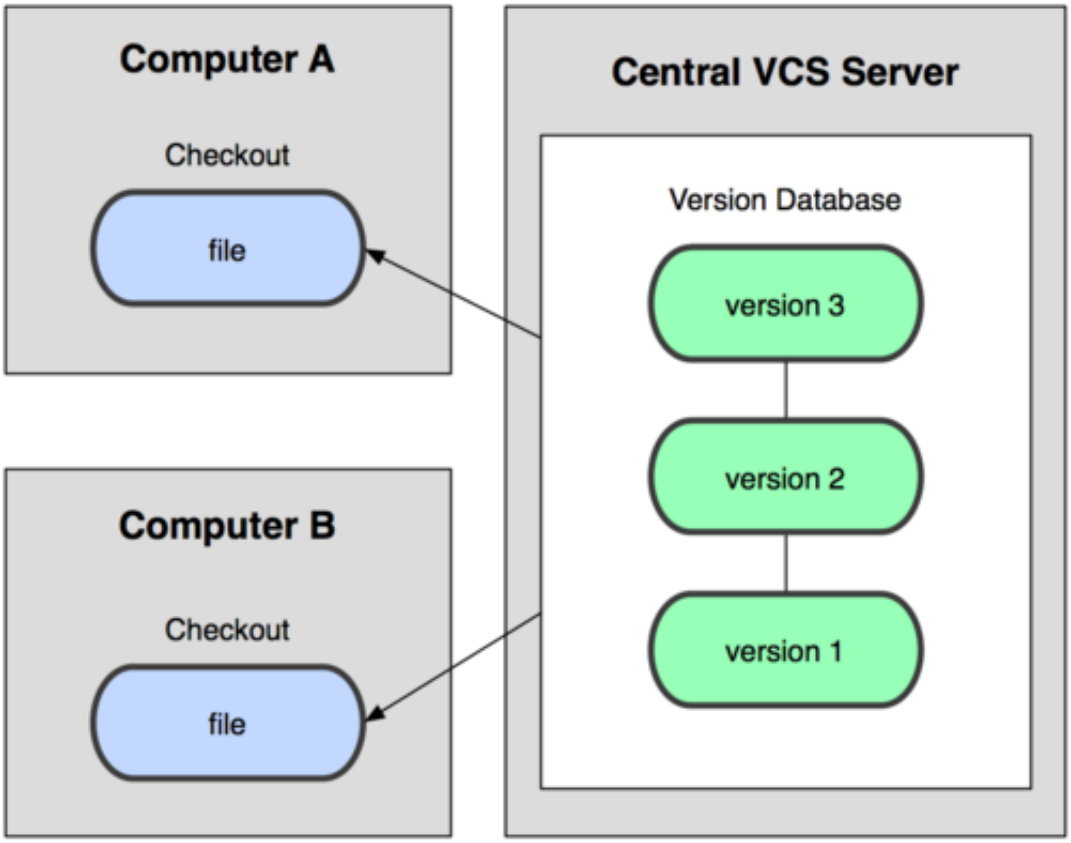
Linus 不到两周时间， C 写了一个分布式版本控制系统，1300 行左右，之后靠 contributors 去壮大。

身世评价：

亲爹：Linus
干爹：世界各地 contributors
外表：Source Tree, TortoiseGit 等等等
内涵：这里

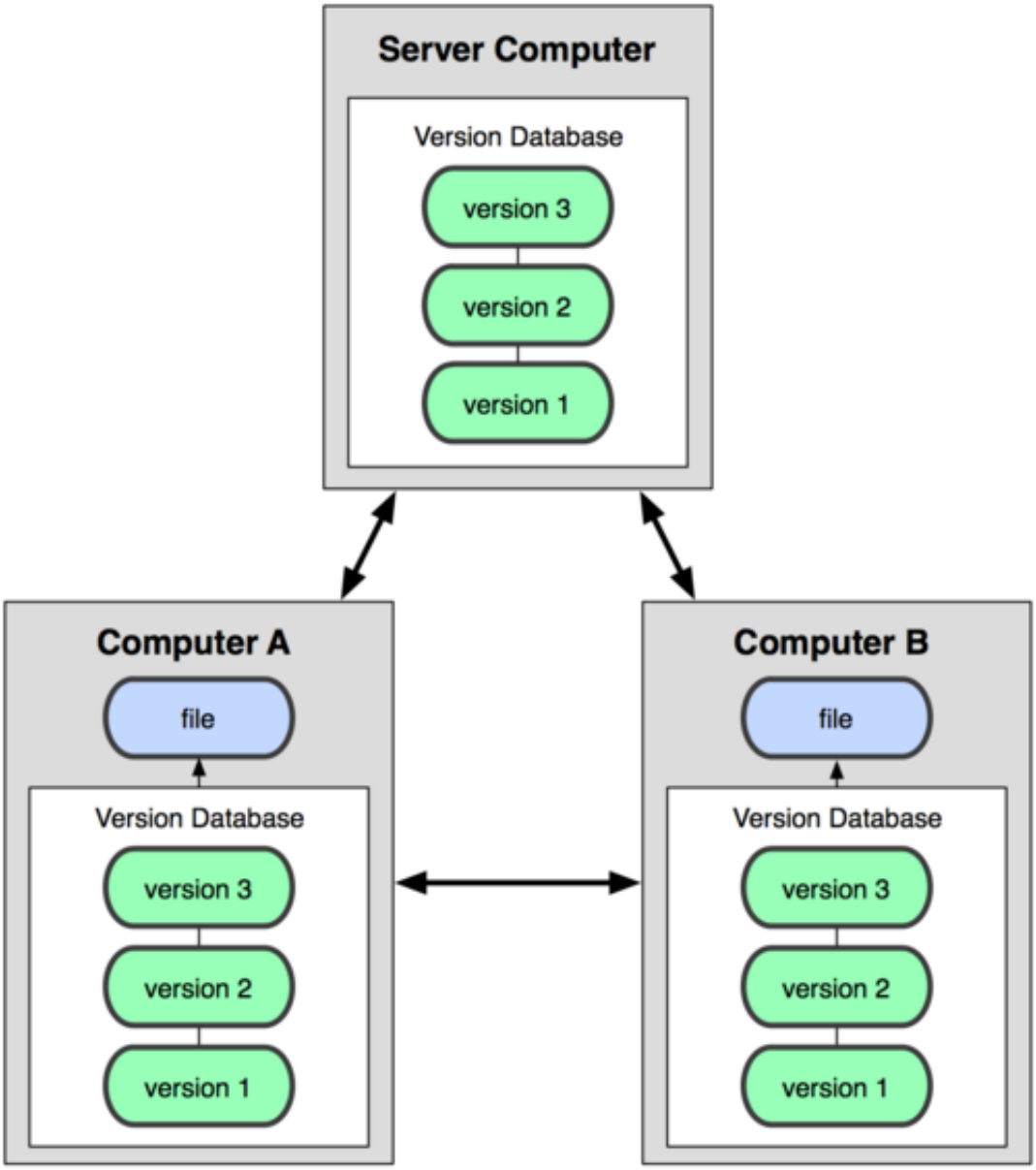
2. 集中式 与 分布式

集中化的版本控制系统：SVN



单一的集中管理的服务器，保存所有文件的修订版本，协同者通过客户端连到这台服务器，查看提交记录或者进行提交。checkout 的只是某个版本的代码，没有任何版本信息记录。

分布式版本控制系统：Git



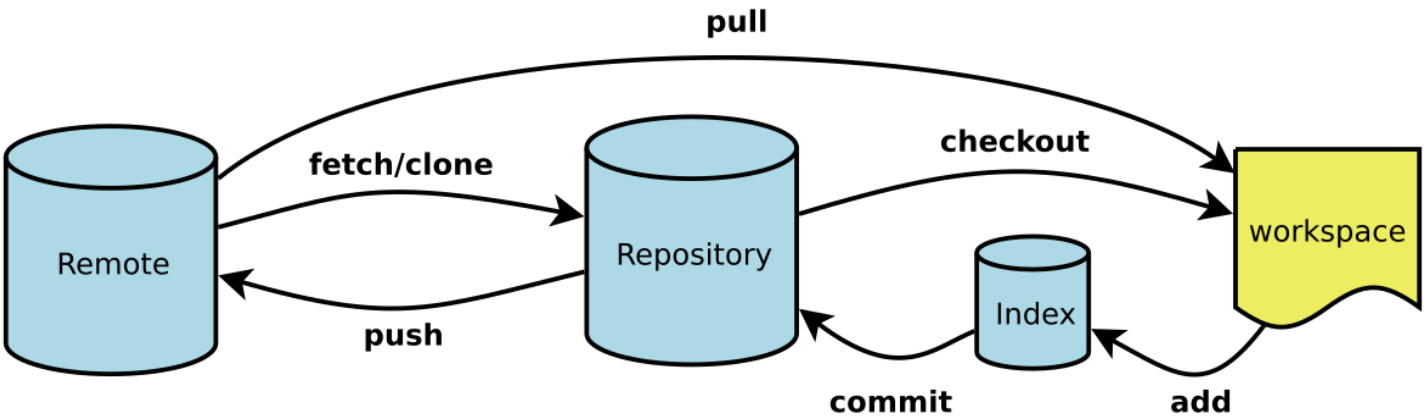
客户端并不只提取最新版本的文件快照，而是把代码仓库 `完整地镜像` 克隆（clone） 提取（fetch) 到本地

- 1.分布式，去中心化
- 2.本地提交
    - 断网提交
    - 小步提交，颗粒化，跟踪代码时，更加细腻
    - 不用 sever，也可以进行版本控制
- 3.高速度，所以 commit，checkout 变得飞快

3. Git 结构模型

为什么要理解 Git 结构模型?

- 我 tmd 在哪?
- 我 tmd 的代码呢?
- 屏幕上的提示信息到底是 tmd 让我干嘛?



两区两库:

- Workspace:工作区,就是你在电脑里能看到的目录
- Index / Stage:暂存区,在暂存区的东西,才能 commit 到 Repository
- Repository:本地仓库
- Remote:远程仓库

六指令:

- add:增加
- commit:提交
- push:推送
- fetch:拉取
- checkout:检出
- pull:fetch + merge

4. Git 命令讲解

1. init clone

```
//  初始化 git
git init
//  从服务器 clone repo,
git clone
```

2. 配置 Git

```
//  查看配置
git config --global --list
//  编辑配置
git config --global --eidt

//  设置提交人
git config --global user.name "John Doe"
//  设置邮件
git config --global user.email johndoe@example.com
//  设置编辑器
git config --global core.editor emacs
```

3. add, stage

从 `工作区` 选取一些代码快照，加入到 `暂存区` ，即将要 `commit` 的内容

```
//  将 <path> 放到 暂存区
git add <path>
//  将 改动的跟踪文件 放到暂存区，但是不包括 新增的
git add -u stages
//  将 所有 放到暂存区
git add .
```

- 提问：为什么会有 `暂存区` 这个概念?
- 快照?

暂存 stash

```
//  暂存当前工作区的变动
git stash
//  暂存当前工作区的变动，并命名为<name>
git save "name"

//  取出暂存，并删除
git stash pop

//  取出暂存，不删除
git stash apply "name"

//  列出暂存列表
git stash list

//  删除暂存
git stash drop "name"
```

高级用法

```
//  操作片段
git add -p

y:  暂存此片
n:  不暂存此片
a:  暂存此片和剩余的片
d:  不暂存和剩余的片
?:  查看帮助
q:  退出
e:  手动编辑选择是否暂存
s:  切片
```

```
git add -i
```

4. commit

将 `暂存区` 的代码快照 `提交` 到 `本地仓库`

```
git commit
//  直接将 工作区 的所有文件提交到`本地仓库`
git commit -a
//  提交暂存区到仓库区
git commit -m "log"
```

高级用法

```
//  纠正最后历史
git commit --amend
//  将 add 新的快照，追加最后到一次提交，并修改log
```

注意：这样会更改历史

5. remote

远端，即远程服务器

```
git remote

//  添加远程主机。
git remote add <主机名> <服务器地址>

//  查看某个远程主机信息
git remote show <主机名>

git remote rm <主机名>
git remote rename <原主机名> <新主机名>
```

6. push

将本地的分支信息推向远端

```
git push <主机名> <本地分支名>:<远程分支名>
```

```
将本地的主干推到远程主机，并建立追踪关系
git push -u origin master:master

//  推送当前分支到追踪分支
git push

//  强制覆盖推送 **改写历史***
git push -f
```

7. pull

pull = fetch + merge pull --rebase = fetch + rebase

merge 优先尝试 fast-forward 模式

```
git pull <远程主机名> <远程分支名>:<本地分支名>
git pull --rebase <远程主机名> <远程分支名>:<本地分支名>
```

8. fetch

将远程仓库新的提交的拉取到本地仓库

```
git fetch
git fetch origin master
```

9. 合并 merge 演合 rebase

LearnGit

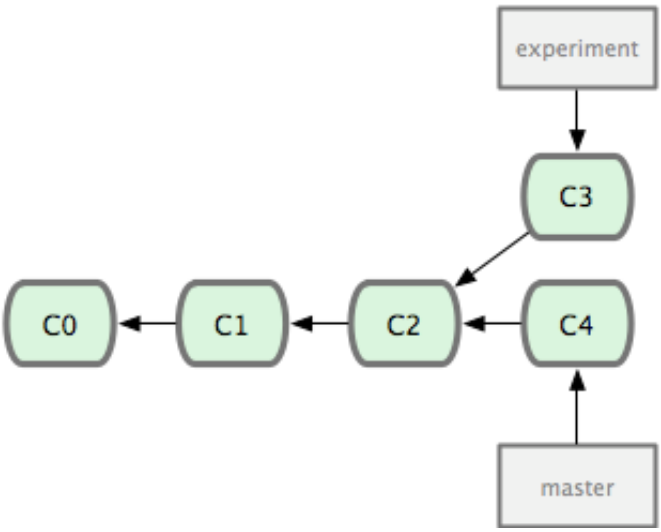
把一个分支中的修改整合到另一个分支的办法有两种：`merge` 和 `rebase`

1. 合并 merge

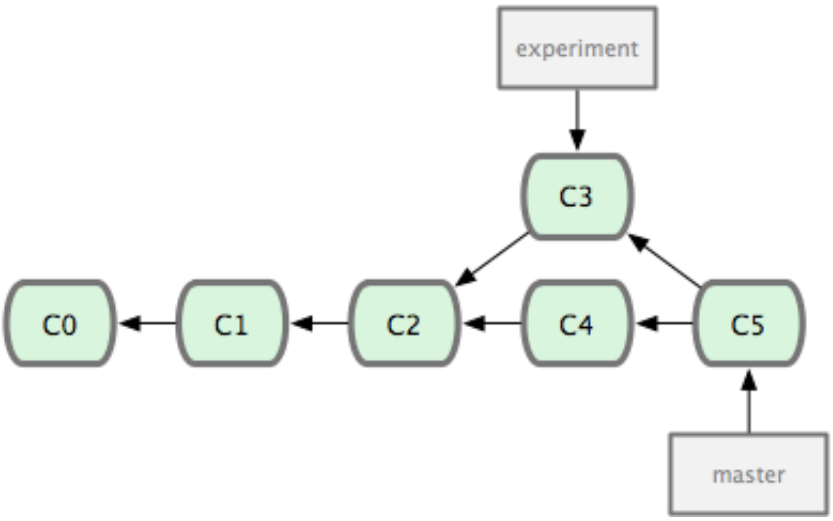
把指定分支 branchX 合并到当前分支，如果不进行 fast-forward 模式，就会产生新的提交点。
若有冲突发生时，新的提交点为解决冲突记录。

```
git merge origin branchX
```

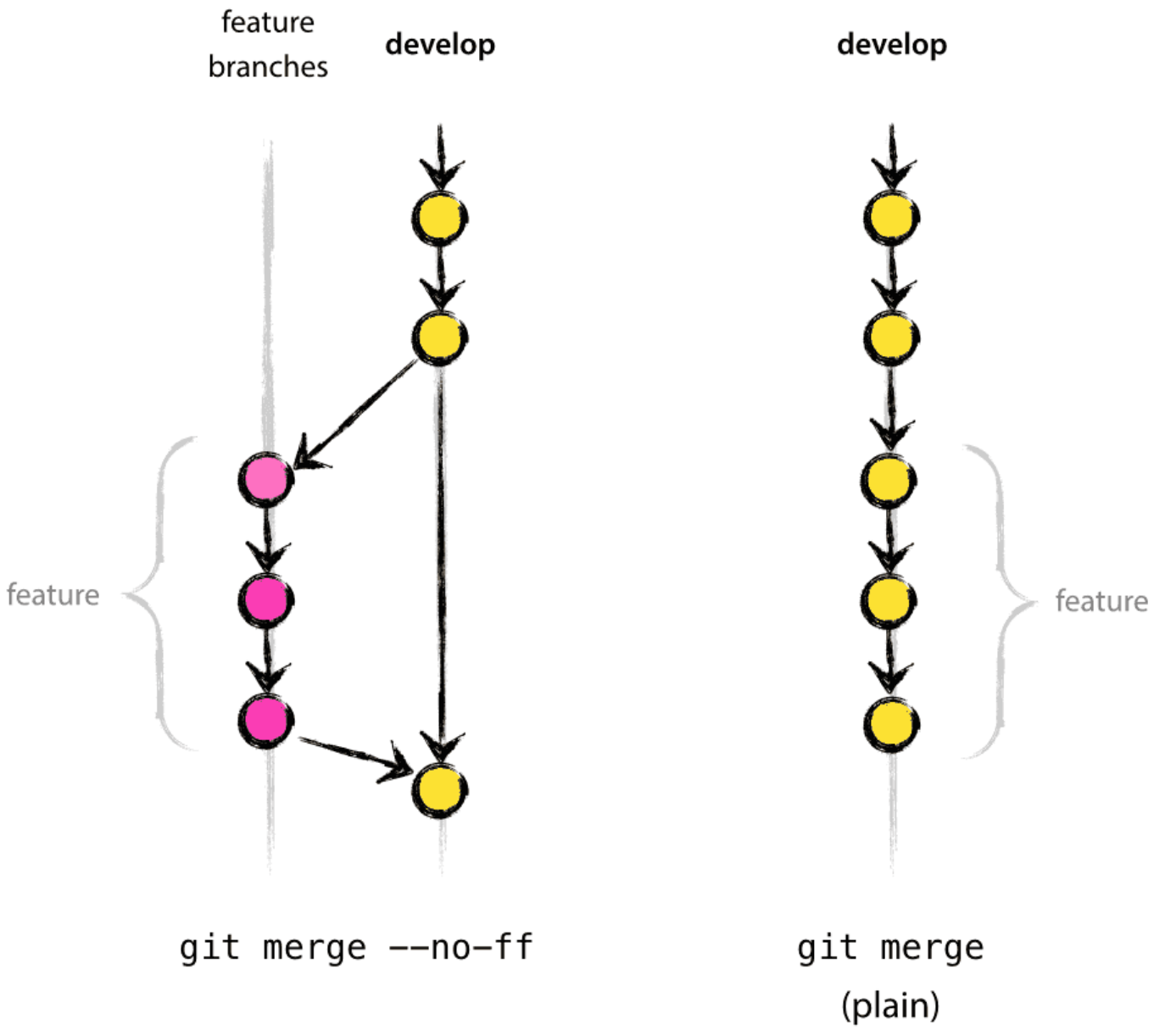
合并前



合并后



fast-forward 模式

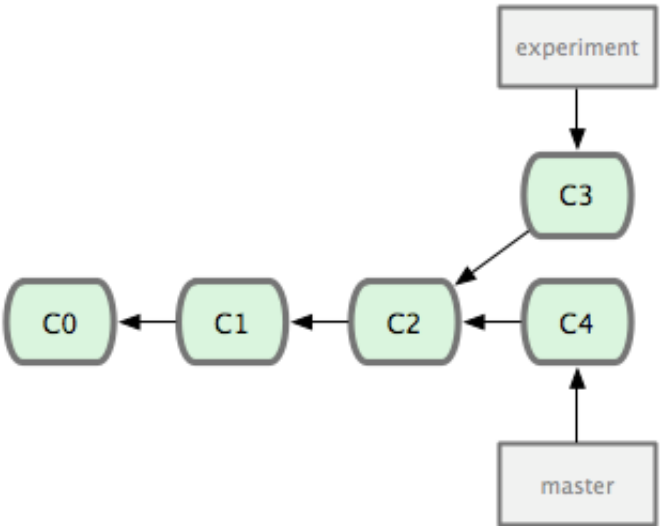


2. 演合 rebase

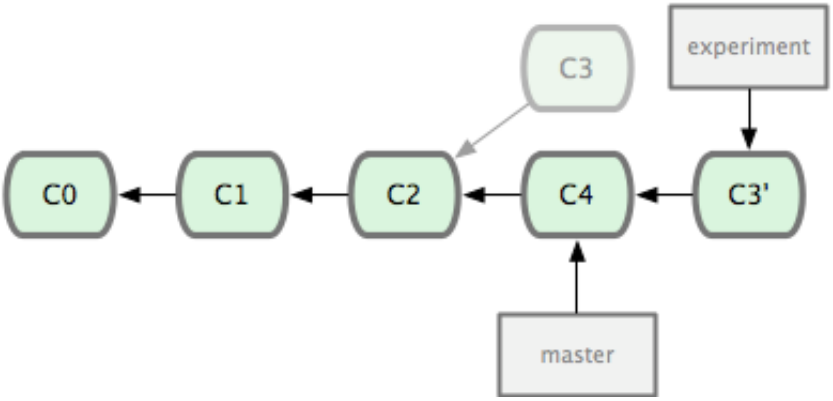
将当前分支和 branchX 产生分歧的 commit 点，重新在 branchX 演一遍。

```
git rebase branchX
```

演合之前



演合之后



注意：永远不要对已经推送到远程仓库的分支进行演合，否则再次推送时会产生冲突。永远不要改变历史。

神奇的演合：

```
git rebase --onto branchA branchB branchC
```

merge 和 rebase 的取舍

rebase: 保证了提交点的干净有序。
merge: 更加详细了记录了开发路线。

10. 后悔药 reset revert reflog

reset

类似 SVN 的revert，将当前分支提交重置回某个提交点。

```
git reset [commit]
//  --soft --mixed --hard
```

注意：不要对已经在远程服务器的 commit 进行 reset

revert

对某一次提交做一次反向操作，并且提交创建一个新提交

```
git revert [commit]
```

reflog

列出 HEAD 经历过的记录，神器~

```
git reflog
```

5. Git 开发模型--GitFlow

1. branch

Git 分支不同于 SVN，不是对文件拷贝的副本，而是快照，使用起来非常轻量级。这使得开发中对分支的 new，merge，delete 变得非常廉价，更好的支持并发型开发。开分支，就是新建一个指针而已。

分支的查看

```
//  列出所有本地分支
git branch

//  列出所有远程分支
git branch -r

//  列出所有分支
git branch -a
```

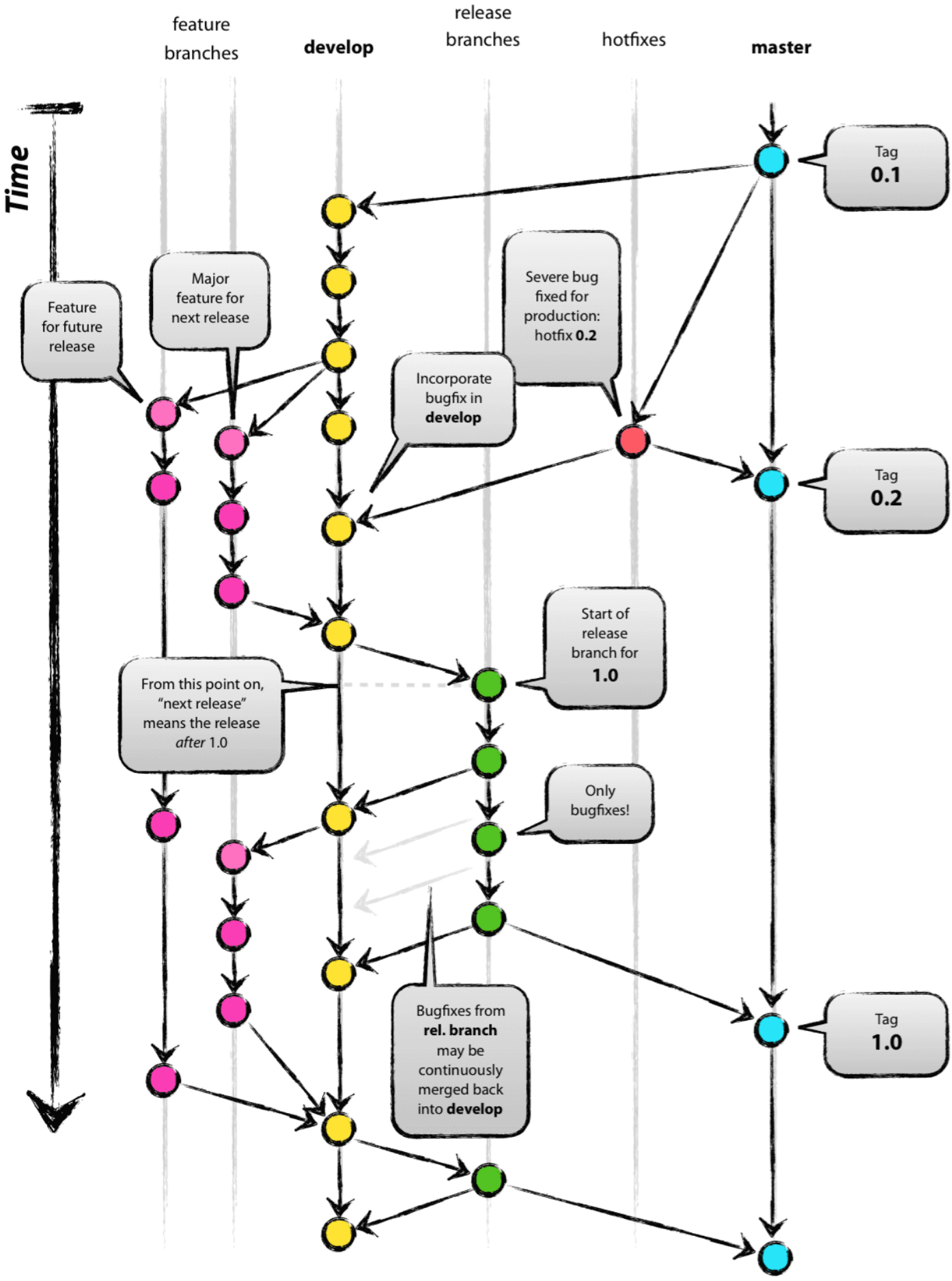
分支的新建

```
//  新建分支 branchName
git branch [branchName]

//  新建分支 branchName 并且切换
git checkout -b [branchName]

//  从一个 commit 点新建一条分支 branchName
git branch [branchName] [commit]
```

分支的切换 checkout 功能

```
//  切换分支
git checkout [branchName]
//  移动 head
git checkout [commit]
//  将追踪的文件重置为上一次 commit 的内容
git checkout <fileName>
```

2. GitFlow


feature branches
develop
release branches
hotfixes
master
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.1
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

6. Git vs SVN

Git 优势

	Git	SVN
代码记录完整性	保留完整的提交记录	合并者会可能会改变提交记录
查看历史版本	checkout 或者 reset； 不依赖网络，速度飞快	revert 或者 checkout； 甚至查看 log 都依赖于网络，速度慢
本地化操作	commit branch stash ...	无
功能并行开发	轻量级分支，创建、切换、合并迅速	checkout 多个工程，多份 copy

SVN 优势

	Git	SVN
提交号	哈希值，基本无意义	增长式提交号，比较版本先后时使用
权限控制	拥有整个代码仓库，随时查看提交记录	基于网络，可设置访问权限

7. 辅助利器

1. Zsh

2. GitDiff

8. 扩展

0. git config

1. git rebase -i

2. cherry-pick

将一个提交点重新应用到当前分支，此时是一个新的提交号

```
git cherry-pick [commit]
```

注意：永远不要 cherry-pick 已推送到远端的 commit，否则再次推送时会产生冲突。

3. .gitignore

4. alias

5. ssh